# 一、常用命令

$ sudo docker # docker 命令帮助

Commands:

attach # 当前 shell 下 attach 连接指定运行镜像

build # 通过 Dockerfile 定制镜像

commit # 提交当前容器为新的镜像

cp # 从容器中拷贝指定文件或者目录到宿主机中

create # 创建一个新的容器，同 run，但不启动容器

diff # 查看 docker 容器变化

events # 从 docker 服务获取容器实时事件

exec # 在已存在的容器上运行命令

export # 导出容器的内容流作为一个 tar 归档文件

[对应 import ]

history # 展示一个镜像形成历史

images # 列出系统当前镜像

import # 从tar包中的内容创建一个新的文件系统映像

[对应 export]

info # 显示系统相关信息

inspect # 查看容器详细信息

kill # kill 指定 docker 容器

load # 从一个 tar 包中加载一个镜像[对应 save]

login # 注册或者登陆一个 docker 源服务器

logout # 从当前 Docker registry 退出

logs # 输出当前容器日志信息

port # 查看映射端口对应的容器内部源端口

pause # 暂停容器

ps # 列出容器列表

pull # 从docker镜像源服务器拉取指定镜像或者库镜像

push # 推送指定镜像或者库镜像至docker源服务器

restart # 重启运行的容器

rm # 移除一个或者多个容器

rmi # 移除一个或多个镜像[无容器使用该镜像才可删除，否则需删除相关容器才可继续或 -f 强制删除]

run # 创建一个新的容器并运行一个命令

save # 保存一个镜像为一个 tar 包[对应 load]

search # 在 docker hub 中搜索镜像

start # 启动容器

stop # 停止容器

tag # 给源中镜像打标签

top # 查看容器中运行的进程信息

unpause # 取消暂停容器

version # 查看 docker 版本号

wait # 截取容器停止时的退出状态值

# 二、Docker 常用命令用法

## 2.1 Search images

$ sudo docker search Ubuntu

示例：

$ sudo docker search --help

Usage: docker search TERM

Search the Docker Hub for images #从 Docker Hub 搜索镜像 --automated=false Only show automated builds

--no-trunc=false Don't truncate output

-s, --stars=0 Only displays with at least xxx stars

$ sudo docker search -s 100 ubuntu

# 查找 star 数至少为 100 的镜像，找出只有官方镜像 start 数超过 100，默认不加 s 选项找出所有相关 ubuntu 镜像

## 2.2 Pull & push images

$ sudo docker pull ubuntu

# 获取 ubuntu 官方镜像

$ sudo docker push

# push 推送指定镜像

$ sudo docker images

# 查看当前镜像列表

Eg:

$ sudo docker pull ubuntu

# 下载官方 ubuntu docker 镜像，默认下载所有 ubuntu 官方库镜像

$ sudo docker pull ubuntu:14.04

# 下载指定版本 ubuntu 官方镜像

$ sudo docker push 192.168.0.100:5000/ubuntu

# 推送镜像库到私有源[可注册 docker 官方账户，推送到官方自有账户]

$ sudo docker push 192.168.0.100:5000/ubuntu:14.04

# 推送指定镜像到私有源

$ sudo docker images

# 显示当前系统镜像，不包括过渡层镜像

$ sudo docker images –a

# 显示当前系统所有镜像，包括过渡层镜像

$ sudo docker images ubuntu

# 显示当前系统 docker ubuntu 库中的所有镜像

## 2.3 Running an interactive shell

$ sudo docker run -i -t ubuntu:14.04 /bin/bash

docker run - 运行一个容器

-t - 分配一个（伪）tty (link is external)

-i - 交互模式 (so we can interact with it)

ubuntu:14.04 - 使用 ubuntu 基础镜像 14.04

/bin/bash - 运行命令 bash shell

注: ubuntu 会有多个版本，通过指定 tag 来启动特定的版本 [image]:[tag]

$ sudo docker ps # 查看当前运行的容器, ps -a 列出当前系统所有的容器

相关快捷键：

退出：Ctrl-Dorexit

detach：Ctrl-P + Ctrl-Q

attach:docker attach CONTAINER-ID

## 2.4 docker info

$ sudo docker info

Containers: 1 # 容器个数

Images: 22 # 镜像个数

Storage Driver: devicemapper # 存储驱动

Pool Name: docker-8:17-3221225728-pool

## 2.5 docker rmi

删除一个或者多个镜像

$ sudo docker rmi --help

-f, --force=false # 强制移除镜像不管是否有容器使用该镜像

--no-prune=false# 不要删除未标记的父镜像

## 2.6 docker run

$ sudo docker run --help

-a, --attach=[] #附加到stdin, stdout or stderr.

-c, --cpu-shares=0 #设置 cpu 使用权重

--cap-add=[] #添加Linux功能

--cap-drop=[] #删除Linux功能

--cidfile="" # 把容器 id 写入到指定文件

--cpuset="" # cpu 绑定 （0-3），（0，1）

-d, --detach=false # 后台运行容器

--device=[] # 将主机装置添加到容器

(e.g. --device=/dev/sdc:/dev/xvdc)

--dns=[] # 设置 dns

--dns-search=[] # 设置 dns 域搜索

-e,--env=[] # 定义环境变量

--entrypoint="" # 覆盖镜像的默认环境变量

--env-file=[] # 从指定文件读取变量值

--expose=[] # 指定对外提供服务端口

-h, --hostname="" # 设置容器主机名

-i, --interactive=false # 保持标准输出开启即使没attached

--link=[] # 添加链接到另外一个容器

--lxc-conf=[] #添加自定义 lxc 选项（eg:--lxc-conf="lxc.cgroup.cpuset.cpus = 0,1" ）

-m, --memory="" # 内存限制

--name="" # 设置容器名

--net="bridge" # 设置容器网络模式

'bridge': #为docker桥上的容器创建一个新的网络栈

'none': # 容器没有网络

'container:<name|id>': #重用另一个网络栈

'host': #使用容器内的主机网络堆栈.

Note: #主机模式给予容器对本地系统服务（例如D-bus）的完全访问，因此被认为是不安全的.

-P, --publish-all=false # 自动映射容器对外提供服务的端口

-p, --publish=[] # 指定端口映射

--privileged=false # 提供更多的权限给容器

--restart="" # 重新启动策略以在容器退出时应用

(no, on-failure[:max-retry], always)

--rm=false # 如果容器退出自动移除和 -d 选项冲突

--security-opt=[] # 安全选项

--sig-proxy=true  # 收到过程中的代理信号， 子进程的退出信号不是代理.

-t, --tty=false # 分配伪终端

-u, --user="" # 指定运行容器的用户 uid 或者用户名

-v, --volume=[] # 挂载卷(从主机: -v /host:/container, 从docker: -v /container)

--volumes-from=[] # 从指定容器挂载卷

-w, --workdir="" # 指定容器工作目录

Eg:

$ sudo docker images ubuntu

$ sudo docker run -t -i -c 100 -m 512MB -h test1 –d --name="docker\_test1" ubuntu /bin/bash

# 创建一个 cpu 优先级为 100，内存限制 512MB，主机名为 test1，名为 docker\_test1 后台运行 bash 的容器 a424ca613c9f2247cd3ede95adfbaf8d28400cbcb1d5f9b69a7b56f97b2b52e5

$ sudo docker ps

pwd /

exit exit

## 2.7 docker start|stop|kill...

dockerstart|stop|kill|restart|pause|unpause|rm|commit|inspect|logs

docker start CONTAINER [CONTAINER...]

# 运行一个或多个停止的容器

docker stop CONTAINER [CONTAINER...]

# 停掉一个或多个运行的容器-t选项可指定超时时间

docker kill [OPTIONS] CONTAINER [CONTAINER...]

# 默认 kill 发送 SIGKILL 信号-s可以指定发送 kill 信号类型

docker restart [OPTIONS] CONTAINER [CONTAINER...]

# 重启一个或多个运行的容器-t选项可指定超时时间

docker pause CONTAINER

# 暂停一个容器，方便 commit

docker unpause CONTAINER

# 继续暂停的容器

docker rm [OPTIONS] CONTAINER [CONTAINER...]

# 移除一个或多个容器

docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]

# 提交指定容器为镜像

-a, --author="" #作者

-m, --message="" #提交信息

-p, --pause=true # 默认 commit 是暂停状态

docker inspect CONTAINER|IMAGE [CONTAINER|IMAGE...]

# 查看容器或者镜像的详细信息

docker logs CONTAINER

# 输出指定容器日志信息

-f, --follow=false # 跟踪日志输出，类似 tail -f

-t, --timestamps=false #显示时间戳

--tail="all" #在日志结尾输出指定的行数(默认为所有日志)

## 2.8 docker exec

docker exec --help

-d, --detach=false #后台运行

-i, --interactive=false #STDIN 保持开放

-t, --tty=false # 分配pseudo-TTY

为了简化调试，可以使用docker exec命令通过 Docker API 和 CLI 在运行的容器上运行程序

$ docker exec -it ubuntu\_bash bash

在容器 ubuntu\_bash 中创建一个新的 Bash 会话

$ docker create -t -i fedora bash

创建了一个可写的容器层 (并且打印出容器 ID)，但是并不运行它，可以使用以下命令运行该容器：

$ docker start -a -i 6d8af538ec5

bash-4.2#

Security Options

通过--security-opt选项，运行容器时用户可自定义 SELinux 和 AppArmor 卷标和配置。

$ docker run --security-opt label:type:svirt\_apache -i -t centos \ bash

只允许容器监听在 Apache 端口，这个选项的好处是用户不需要运行 docker 的时候指定--privileged选项，降低安全风险。

## 2.9 Docker 端口映射

1. 自动映射端口

-P使用时需要指定--expose选项，指定需要对外提供服务的端口

$ sudo docker run -t -P --expose 22 --name server ubuntu:14.04

使用docker run -P自动绑定所有对外提供服务的容器端口，映射的端口将会从没有使用的端口池中 (49000..49900) 自动选择，你可以通过docker ps、docker inspect <container\_id>或者docker port <container\_id> <port>确定具体的绑定信息

1. 绑定端口到指定接口

$ sudo docker run -p [([<host\_interface>:[host\_port]])|(<host\_port>):]<container\_port>[/udp] <image> <cmd>

默认不指定绑定 ip 则监听所有网络接口。

1. 绑定 TCP 端口

# 将容器的TCP端口8080绑定到主机的127.0.0.1上的TCP端口80

$ sudo docker run -p 127.0.0.1:80:8080 <image> <cmd>

# 将容器的TCP端口8080绑定到主机的127.0.0.1上的动态分配的TCP端口

$ sudo docker run -p 127.0.0.1::8080 <image> <cmd>

# 将容器的TCP端口8080绑定到主机所有可用接口上的TCP端口80.

$ sudo docker run -p 80:8080 <image> <cmd>

# 将容器的TCP端口8080绑定到所有可用接口上的动态分配的TCP端口

$ sudo docker run -p 8080 <image> <cmd>

1. 绑定 UDP 端口

# 将容器的UDP端口5353绑定到主机的127.0.0.1上的UDP端口53

$ sudo docker run -p 127.0.0.1:53:5353/udp <image> <cmd>

# 三、命名空间「Namespaces」

可看博客（<http://www.infoq.com/cn/articles/docker-kernel-knowledge-namespace-resource-isolation?utm_source=tuicool>）认识namespace

## 3.1 pid namespace

不同用户的进程就是通过 pid namespace 隔离开的，且不同 namespace 中可以有相同 PID。具有以下特征:

* 每个 namespace 中的 pid 是有自己的 pid=1 的进程(类似 /sbin/init 进程)
* 每个 namespace 中的进程只能影响自己的同一个 namespace 或子 namespace 中的进程
* 因为 /proc 包含正在运行的进程，因此在 container 中的 pseudo-filesystem 的 /proc 目录只能看到自己 namespace 中的进程
* 因为 namespace 允许嵌套，父 namespace 可以影响子 namespace 的进程，所以子 namespace 的进程可以在父 namespace 中看到，但是具有不同的pid

## 3.2 mnt namespace

类似 chroot，将一个进程放到一个特定的目录执行。mnt namespace 允许不同 namespace 的进程看到的文件结构不同，这样每个 namespace 中的进程所看到的文件目录就被隔离开了。同 chroot 不同，每个 namespace 中的 [Container](http://lib.csdn.net/base/docker) 在 /proc/mounts 的信息只包含所在 namespace 的 mount point。

## 3.3 net namespace

网络隔离是通过 net namespace 实现的， 每个 net namespace 有独立的 network devices, IP addresses, IP routing tables, /proc/net 目录。这样每个 container 的网络就能隔离开来。 docker 默认采用 veth 的方式将 container 中的虚拟网卡同 host 上的一个 docker bridge 连接在一起。

## 3.4 uts namespace

UTS ("UNIX Time-sharing System") namespace 允许每个 container 拥有独立的 hostname 和 domain name, 使其在网络上可以被视作一个独立的节点而非 Host 上的一个进程。

## 3.5 ipc namespace

container 中进程交互还是采用 Linux 常见的进程间交互方法 (interprocess communication - IPC), 包括常见的信号量、消息队列和共享内存。然而同 VM 不同，container 的进程间交互实际上还是 host 上具有相同 pid namespace 中的进程间交互，因此需要在IPC资源申请时加入 namespace 信息 - 每个 IPC 资源有一个唯一的 32bit ID。

## 3.6 user namespace

每个 container 可以有不同的 user 和 group id, 也就是说可以以 container 内部的用户在 container 内部执行程序而非 Host 上的用户。

# 四、Docker 四种网络模式

docker run 创建 Docker 容器时，可以用 --net 选项指定容器的网络模式，Docker 有以下 4 种网络模式：

* host 模式，使用 --net=host 指定。
* container 模式，使用 --net=container:NAMEorID 指定。
* none 模式，使用 --net=none 指定。
* bridge 模式，使用 --net=bridge 指定，默认设置。

## 4.1 host模式

如果启动容器的时候使用 host 模式，那么这个容器将不会获得一个独立的 Network Namespace，而是和宿主机共用一个 Network Namespace。容器将不会虚拟出自己的网卡，配置自己的 IP 等，而是使用宿主机的 IP 和端口。

例如，在 192.168.15.216/24 的机器上用 host 模式启动一个含有 web 应用的 Docker 容器，监听 tcp 80 端口。当在容器中执行任何类似 ifconfig 命令查看网络环境时，看到的都是宿主机上的信息。而外界访问容器中的应用，则直接使用 192.168.15.216:80 即可，不用任何 NAT 转换，就如直接跑在宿主机中一样。但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。如digits的访问方式

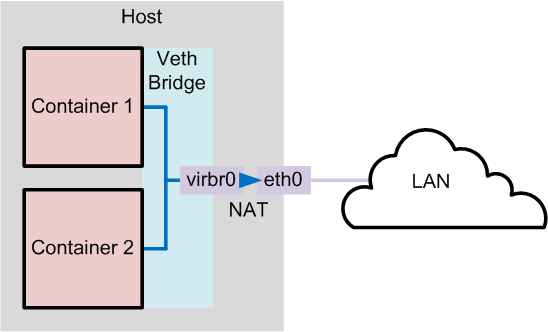
## 4.2 container 模式

这个模式指定新创建的容器和已经存在的一个容器共享一个 Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的 IP，而是和一个指定的容器共享 IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过 lo 网卡设备通信。（lo 本地环路接口 <http://soft.chinabyte.com/os/327/11628327.shtml>）

## 4.3 none模式

这个模式和前两个不同。在这种模式下，Docker 容器拥有自己的 Network Namespace，但是，并不为 Docker容器进行任何网络配置。也就是说，这个 Docker 容器没有网卡、IP、路由等信息。需要我们自己为 Docker 容器添加网卡、配置 IP 等。

## 4.4 bridge模式



bridge 模式是 Docker 默认的网络设置，此模式会为每一个容器分配 Network Namespace、设置 IP 等，并将一个主机上的 Docker 容器连接到一个虚拟网桥上。当 Docker server 启动时，会在主机上创建一个名为 docker0 的虚拟网桥，此主机上启动的 Docker 容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。接下来就要为容器分配 IP 了，Docker 会从 RFC1918 所定义的私有 IP 网段中，选择一个和宿主机不同的IP地址和子网分配给 docker0，连接到 docker0 的容器就从这个子网中选择一个未占用的 IP 使用。如一般 Docker 会使用 172.17.0.0/16 这个网段，并将 172.17.42.1/16 分配给 docker0 网桥（在主机上使用 ifconfig 命令是可以看到 docker0 的，可以认为它是网桥的管理接口，在宿主机上作为一块虚拟网卡使用）

可参考（<http://www.wickedawesometech.us/2014/07/the-container-world-part-2-networking.html>）

## 4.5 网络常用命令

1. 列出当前主机网桥

$ sudo brctl show

# brctl 工具依赖 bridge-utils 软件包

1. 查看当前 docker0 ip

$ sudo ifconfig docker0

在容器运行时，每个容器都会分配一个特定的虚拟机口并桥接到 docker0。每个容器都会配置同 docker0 ip 相同网段的专用 ip 地址，docker0 的 IP 地址被用于所有容器的默认网关。

1. 运行一个容器

$ sudo docker run -t -i -d ubuntu /bin/bash

$ sudo brctl show

以上, docker0 扮演着容器的虚拟接口桥接的角色。

1. 使用特定范围的 IP

Docker 会尝试寻找没有被主机使用的 ip 段，尽管它适用于大多数情况下，但是它不是万能的，有时候我们还是需要对 ip 进一步规划。Docker 允许你管理 docker0 桥接或者通过-b选项自定义桥接网卡，需要安装bridge-utils软件包。

基本步骤如下：

* ensure Docker is stopped

# 确保 docker 的进程是停止的

* create your own bridge (bridge0 for example)

# 创建自定义网桥

* assign a specific IP to this bridge

# 给网桥分配特定的 ip

* start Docker with the -b=bridge0 parameter

# 以 -b 的方式指定网桥

Eg:

# Stopping Docker and removing docker0

$ sudo service docker stop

$ sudo ip link set dev docker0 down

$ sudo brctl delbr docker0

# Create our own bridge

$ sudo brctl addbr bridge0

$ sudo ip addr add 192.168.5.1/24 dev bridge0

$ sudo ip link set dev bridge0 up

# Confirming that our bridge is up and running

$ ip addr show bridge0

# Tell Docker about it and restart (on Ubuntu)

$ echo 'DOCKER\_OPTS="-b=bridge0"'

>> /etc/default/docker

$ sudo service docker start

## 4.6 不同主机间容器通信

不同容器之间的通信可以借助于 pipework 这个工具：

$ git clone https://github.com/jpetazzo/pipework.git

$ sudo cp -rp pipework/pipework /usr/local/bin/

安装相应依赖软件

$ sudo apt-get install iputils-arping bridge-utils -y

# brctl show

bridge name bridge id STP enabled interfaces

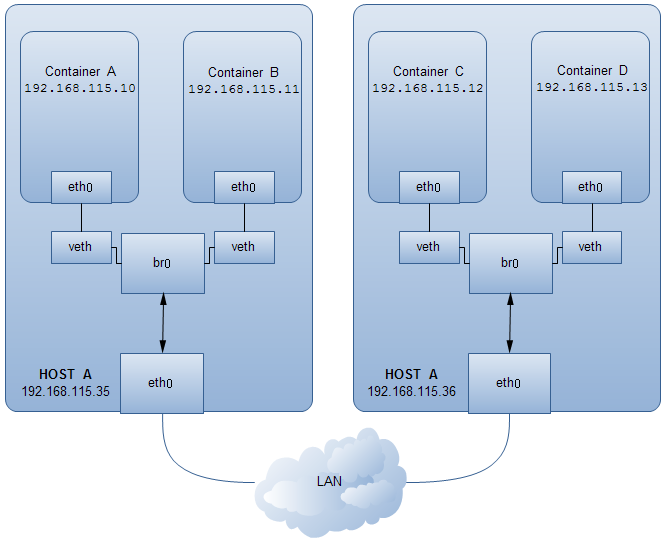
br0 8000.000c291412cd no eth0

docker0 8000.56847afe9799 no vetheb48029

可以删除 docker0，直接把 docker 的桥接指定为 br0。也可以保留使用默认的配置，这样单主机容器之间的通信可以通过 docker0，而跨主机不同容器之间通过 pipework 新建 docker 容器的网卡桥接到 br0，这样跨主机容器之间就可以通信了。

* ubuntu
* $ sudo service docker stop
* $ sudo ip link set dev docker0 down
* $ sudo brctl delbr docker0
* $ echo 'DOCKER\_OPTS="-b=br0"' >> /etc/default/docker
* $ sudo service docker start
* CentOS 7/RHEL 7
* $ sudo systemctl stop docker
* $ sudo ip link set dev docker0 down
* $ sudo brctl delbr docker0
* $ cat /etc/sysconfig/docker | grep 'OPTIONS='OPTIONS=--selinux-enabled -b=br0 -H fd://
* $ sudo systemctl start docker

Pipework



不同容器之间的通信可以借助于 pipework 这个工具给 docker 容器新建虚拟网卡并绑定 IP 桥接到 br0

$ git clone https://github.com/jpetazzo/pipework.git

$ sudo cp -rp pipework/pipework /usr/local/bin/

$ pipework

如果删除了默认的 docker0 桥接，把 docker 默认桥接指定到了 br0，则最好在创建容器的时候加上--net=none，防止自动分配的 IP 在局域网中有冲突。

Eg:

$ sudo docker run --rm -ti --net=none ubuntu:14.04 /bin/bash

root@a46657528059:/#

$ # Ctrl-P + Ctrl-Q 回到宿主机 shell，容器 detach 状态

$ sudo docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

a46657528059 ubuntu:14.04 "/bin/bash" 4 minutes ago Up 4 minutes hungry\_lalande

$ sudo pipework br0 -i eth0 a46657528059 192.168.115.10/24@192.168.115.2

# 默认不指定网卡设备名，则默认添加为 eth1

# 另外 pipework 不能添加静态路由，如果有需求则可以在 run 的时候加上 --privileged=true 权限在容器中手动添加，

# 但这种安全性有缺陷，可以通过 ip netns 操作

$ sudo docker attach a46657528059

$ ifconfig eth0

eth0 Link encap:Ethernet HWaddr 86:b6:6b:e8:2e:4d

inet addr:192.168.115.10 Bcast:0.0.0.0 Mask:255.255.255.0

inet6 addr: fe80::84b6:6bff:fee8:2e4d/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:8 errors:0 dropped:0 overruns:0 frame:0

TX packets:9 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:648 (648.0 B) TX bytes:690 (690.0 B)

$ route -n

Kernel IP routing table

Destination Gateway Genmask Flags Metric Ref Use Iface

0.0.0.0 192.168.115.2 0.0.0.0 UG 0 0 0 eth0

192.168.115.0 0.0.0.0 255.255.255.0 U 0 0 0 eth0

使用ip netns添加静态路由，避免创建容器使用--privileged=true选项造成一些不必要的安全问题：

$ docker inspect --format="{{ .State.Pid }}" a46657528059 # 获取指定容器 pid6350

$ sudo ln -s /proc/6350/ns/net /var/run/netns/6350

$ sudo ip netns exec 6350 ip route add 192.168.0.0/16 dev eth0 via 192.168.115.2

$ sudo ip netns exec 6350 ip route # 添加成功

192.168.0.0/16 via 192.168.115.2 dev eth0

可参考（1、<http://blog.sina.com.cn/s/blog_524394da0102v9jd.html>

2、<https://github.com/jpetazzo/pipework>

3、<http://www.infoq.com/cn/articles/docker-network-and-pipework-open-source-explanation-practice>）

# 五、Dockerfile

Docker 可以通过 Dockerfile 的内容来自动构建镜像。Dockerfile 是一个包含创建镜像所有命令的文本文件，通过docker build命令可以根据 Dockerfile 的内容构建镜像，在介绍如何构建之前先介绍下 Dockerfile 的基本语法结构。

Dockerfile 有以下指令选项:

* FROM
* MAINTAINER
* RUN
* CMD
* EXPOSE
* ENV
* ADD
* COPY
* ENTRYPOINT
* VOLUME
* USER
* WORKDIR
* ONBUILD

## 5.1 FROM

用法:

FROM <image>

* FROM指定构建镜像的基础源镜像，如果本地没有指定的镜像，则会自动从 Docker 的公共库 pull 镜像下来。
* FROM必须是 Dockerfile 中非注释行的第一个指令，即一个 Dockerfile 从FROM语句开始。
* FROM可以在一个 Dockerfile 中出现多次，如果有需求在一个 Dockerfile 中创建多个镜像。
* 如果FROM语句没有指定镜像标签，则默认使用latest标签。

## 5.2 MAINTAINER

用法:

MAINTAINER <name>

指定创建镜像的用户

RUN 有两种使用方式：

1. RUN
2. RUN ["executable", "param1", "param2"](http://opskumu.github.io/exec%20form)

每条RUN指令将在当前镜像基础上执行指定命令，并提交为新的镜像，后续的RUN都在之前RUN提交后的镜像为基础，镜像是分层的，可以通过一个镜像的任何一个历史提交点来创建，类似源码的[版本控制](http://lib.csdn.net/base/git)。

exec 方式会被解析为一个 JSON 数组，所以必须使用双引号而不是单引号。exec 方式不会调用一个命令 shell，所以也就不会继承相应的变量，如：

RUN [ "echo", "$HOME" ]

这种方式是不会达到输出 HOME 变量的，正确的方式应该是这样的

RUN [ "sh", "-c", "echo", "$HOME" ]

RUN产生的缓存在下一次构建的时候是不会失效的，会被重用，可以使用--no-cache选项，即docker build --no-cache，如此便不会缓存。

## 5.3 CMD

CMD有三种使用方式:

1. CMD ["executable","param1","param2"](http://opskumu.github.io/exec%20form,%20this%20is%20the%20preferred%20form,%20%E4%BC%98%E5%85%88%E9%80%89%E6%8B%A9)
2. CMD ["param1","param2"](http://opskumu.github.io/as%20default%20parameters%20to%20%60ENTRYPOINT%60)
3. CMD command param1 param2 (shell form)

* CMD指定在 Dockerfile 中只能使用一次，如果有多个，则只有最后一个会生效。
* CMD的目的是为了在启动容器时提供一个默认的命令执行选项。如果用户启动容器时指定了运行的命令，则会覆盖掉CMD指定的命令。
* CMD会在启动容器的时候执行，build 时不执行，而RUN只是在构建镜像的时候执行，后续镜像构建完成之后，启动容器就与RUN无关了。

## 5.4 EXPOSE

用法：

EXPOSE <port> [<port>...]

告诉 Docker 服务端容器对外映射的本地端口，需要在 docker run 的时候使用-p或者-P选项生效。

## 5.5 ENV

用法：

ENV <key> <value> # 只能设置一个变量

ENV <key>=<value> ... # 允许一次设置多个变量

指定一个环节变量，会被后续RUN指令使用，并在容器运行时保留。

例子:

ENV myName="John Doe" myDog=Rex\ The\ Dog \

myCat=fluffy

等同于

ENV myName John Doe

ENV myDog Rex The Dog

ENV myCat fluffy

## 5.6 ADD

用法：

ADD <src>... <dest>

ADD复制本地主机文件、目录或者远程文件 URLS 从 并且添加到容器指定路径中 。

ADD hom\* /mydir/

ADD hom?.txt /mydir/

* 路径如果不存在，会自动创建对应目录
* 路径必须是 Dockerfile 所在路径的相对路径
* 如果是一个目录，只会复制目录下的内容，而目录本身则不会被复制

## 5.7 COPY

用法：

COPY <src>... <dest>

COPY复制新文件或者目录从 并且添加到容器指定路径中 。用法同ADD，唯一的不同是不能指定远程文件 URLS。

## 5.8 ENTRYPOINT

用法：

1. ENTRYPOINT ["executable", "param1", "param2"](http://opskumu.github.io/the%20preferred%20exec%20form%EF%BC%8C%E4%BC%98%E5%85%88%E9%80%89%E6%8B%A9)
2. ENTRYPOINT command param1 param2 (shell form)

配置容器启动后执行的命令，并且不可被 docker run 提供的参数覆盖，而CMD是可以被覆盖的。如果需要覆盖，则可以使用docker run --entrypoint选项。

每个 Dockerfile 中只能有一个ENTRYPOINT，当指定多个时，只有最后一个生效。

Eg:

通过ENTRYPOINT使用 exec form 方式设置稳定的默认命令和选项，而使用CMD添加默认之外经常被改动的选项。

FROM ubuntu

ENTRYPOINT ["top", "-b"]

CMD ["-c"]

通过 Dockerfile 使用ENTRYPOINT展示前台运行 Apache 服务

FROM debian:stable

RUN apt-get update && apt-get install -y --force-yes apache2

EXPOSE 80 443

VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]

ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]

Eg:

使用Shell form ENTRYPOINT方式会在/bin/sh -c中执行，会忽略任何CMD或者docker run命令行选项，为了确保docker stop能够停止长时间运行ENTRYPOINT的容器，确保执行的时候使用exec选项。

FROM ubuntu

ENTRYPOINT exec top -b

如果在ENTRYPOINT忘记使用exec选项，则可以使用CMD补上:

FROM ubuntu

ENTRYPOINT top -b

CMD --ignored-param1 # --ignored-param2 ... --ignored-param3 ...

## 5.9 VOLUME

用法：

VOLUME ["/data"]

创建一个可以从本地主机或其他容器挂载的挂载点。

## 5.10 USER

用法：

USER daemon

指定运行容器时的用户名或 UID，后续的RUN、CMD、ENTRYPOINT也会使用指定用户。

## 5.11 WORKDIR

用法：

WORKDIR /path/to/workdir

为后续的RUN、CMD、ENTRYPOINT指令配置工作目录。可以使用多个WORKDIR指令，后续命令如果参数是相对路径，则会基于之前命令指定的路径。

WORKDIR /a

WORKDIR b

WORKDIR c

RUN pwd

最终路径是/a/b/c。

WORKDIR指令可以在ENV设置变量之后调用环境变量:

ENV DIRPATH /path

WORKDIR $DIRPATH/$DIRNAME

最终路径则为 /path/$DIRNAME。

## 5.12 ONBUILD

用法：

ONBUILD [INSTRUCTION]

配置当所创建的镜像作为其它新创建镜像的基础镜像时，所执行的操作指令。

例如，Dockerfile 使用如下的内容创建了镜像 image-A：

[...]

ONBUILD ADD . /app/src

ONBUILD RUN /usr/local/bin/python-build --dir /app/src

[...]

如果基于 image-A 创建新的镜像时，新的 Dockerfile 中使用 FROM image-A 指定基础镜像时，会自动执行 ONBUILD 指令内容，等价于在后面添加了两条指令。

# Automatically run the following

ADD . /app/src

RUN /usr/local/bin/python-build --dir /app/src

使用ONBUILD指令的镜像，推荐在标签中注明，例如 ruby:1.9-onbuild。

## 5.13 example

示例：

# Nginx

#

# VERSION 0.0.1

FROM ubuntu

MAINTAINER Victor Vieux <victor@docker.com>

RUN apt-get update && apt-get install -y inotify-tools nginx apache2 openssh-server

# Firefox over VNC

#

# VERSION 0.3

FROM ubuntu

# Install vnc, xvfb in order to create a 'fake' display and firefox

RUN apt-get update && apt-get install -y x11vnc xvfb firefox

RUN mkdir ~/.vnc

# Setup a password

RUN x11vnc -storepasswd 1234 ~/.vnc/passwd

# Autostart firefox (might not be the best way, but it does the trick)

RUN bash -c 'echo "firefox" >> /.bashrc'

EXPOSE 5900

CMD ["x11vnc", "-forever", "-usepw", "-create"]

# Multiple images example

#

# VERSION 0.1

FROM ubuntu

RUN echo foo > bar

# Will output something like ===> 907ad6c2736f

FROM ubuntu

RUN echo moo > oink

# Will output something like ===> 695d7793cbe4

# You᾿ll now have two images, 907ad6c2736f with /bar, and 695d7793cbe4 with

# /oink.

## 5.14 docker build

示例：

$ docker build --help

#从源代码中构建一个新的镜像

--force-rm=false # 移除过渡容器，即使构建失败

--no-cache=false # 构建镜像不实用 cache

-q, --quiet=false # 控制容器生成冗长的输出

--rm=true # 构建成功后移除过渡层容器

-t, --tag="" # 在成功的情况下应用于生成的镜像的存储库名称（以及可选的标记）

## 5.15 dockerfile实践

* 使用.dockerignore文件

为了在docker build过程中更快上传和更加高效，应该使用一个.dockerignore文件用来排除构建镜像时不需要的文件或目录。例如,除非.[Git](http://lib.csdn.net/base/git)在构建过程中需要用到，否则你应该将它添加到.dockerignore文件中，这样可以节省很多时间。

* 避免安装不必要的软件包

为了降低复杂性、依赖性、文件大小以及构建时间，应该避免安装额外的或不必要的包。例如，不需要在一个[数据库](http://lib.csdn.net/base/mysql)镜像中安装一个文本编辑器。

* 每个容器都跑一个进程

在大多数情况下，一个容器应该只单独跑一个程序。解耦应用到多个容器使其更容易横向扩展和重用。如果一个服务依赖另外一个服务，可以参考 [Linking Containers Together](https://docs.docker.com/userguide/dockerlinks/)。

* 最小化层

我们知道每执行一个指令，都会有一次镜像的提交，镜像是分层的结构，对于Dockerfile，应该找到可读性和最小化层之间的平衡。

* 多行参数排序

如果可能，通过字母顺序来排序，这样可以避免安装包的重复并且更容易更新列表，另外可读性也会更强，添加一个空行使用\换行:

RUN apt-get update && apt-get install -y \

bzr \

cvs \

git \

mercurial \

subversion

* 创建缓存

镜像构建过程中会按照Dockerfile的顺序依次执行，每执行一次指令 Docker 会寻找是否有存在的镜像缓存可复用，如果没有则创建新的镜像。如果不想使用缓存，则可以在docker build时添加--no-cache=true选项。

## 5.16 Dockerfile 指令

FROM: 只要可能就使用官方镜像库作为基础镜像

RUN: 为保持可读性、方便理解、可维护性，把长或者复杂的RUN语句使用\分隔符分成多行

不建议RUN apt-get update独立成行，否则如果后续包有更新，那么也不会再执行更新

避免使用RUN apt-get upgrade或者dist-upgrade，很多必要的包在一个非privileged权限的容器里是无法升级的。如果知道某个包更新，使用apt-get install -y xxx

标准写法

RUN apt-get update && apt-get install -y package-bar package-foo

例子:

RUN apt-get update && apt-get install -y \

aufs-tools \

automake \

btrfs-tools \

build-essential \

curl \

dpkg-sig \

git \

iptables \

libapparmor-dev \

libcap-dev \

libsqlite3-dev \

lxc=1.0\* \

mercurial \

parallel \

reprepro \

ruby1.9.1 \

ruby1.9.1-dev \

s3cmd=1.1.0\*

CMD: 推荐使用CMD [“executable”, “param1”, “param2”…]这种格式，CMD [“param”, “param”]则配合ENTRYPOINT使用

EXPOSE: Dockerfile 指定要公开的端口，使用docker run时指定映射到宿主机的端口即可

ENV: 为了使新的软件更容易运行，可以使用ENV更新PATH变量。如ENV PATH /usr/local/nginx/bin:$PATH确保CMD ["nginx"]即可运行

ENV也可以这样定义变量：

ENV PG\_MAJOR 9.3

ENV PG\_VERSION 9.3.4

RUN curl -SL http://example.com/postgres-$PG\_VERSION.tar.xz | tar -xJC /usr/src/postgress && …

ENV PATH /usr/local/postgres-$PG\_MAJOR/bin:$PATH

ADDorCOPY:ADD比COPY多一些特性「tar 文件自动解包和支持远程 URL」，不推荐添加远程 URL

如不推荐这种方式:

ADD http://example.com/big.tar.xz /usr/src/things/

RUN tar -xJf /usr/src/things/big.tar.xz -C /usr/src/things

RUN make -C /usr/src/things all

推荐使用 curl 或者 wget 替换，使用如下方式:

RUN mkdir -p /usr/src/things \

&& curl -SL http://example.com/big.tar.gz \

| tar -xJC /usr/src/things \

&& make -C /usr/src/things all

如果不需要添加 tar 文件，推荐使用COPY。

# 六、容器数据管理

docker管理数据的方式有两种：

* 数据卷
* 数据卷容器

## 6.1 数据卷

数据卷是一个或多个容器专门指定绕过Union File System的目录，为持续性或共享数据提供一些有用的功能：

* 数据卷可以在容器间共享和重用
* 数据卷数据改变是直接修改的
* 数据卷数据改变不会被包括在容器中
* 数据卷是持续性的，直到没有容器使用它们

1. 添加一个数据卷：

你可以使用-v选项添加一个数据卷，或者可以使用多次-v选项为一个 docker 容器运行挂载多个数据卷。

$ sudo docker run --name data -v /data -t -i ubuntu:14.04 /bin/bash

# 创建数据卷绑定到到新建容器，新建容器中会创建 /data 数据卷

bash-4.1# ls -ld /data/

drwxr-xr-x 2 root root 4096 Jul 23 06:59 /data/

bash-4.1# df -Th

Filesystem Type Size Used Avail Use% Mounted on

... ...

ext4 91G 4.6G 82G 6% /data

1. **创建的数据卷可以通过docker inspect获取宿主机对应路径**

**$ sudo docker inspect data**

**... ... "Volumes": { "/data": "/var/lib/docker/vfs/dir/151de401d268226f96d824fdf444e77a4500aed74c495de5980c807a2ffb7ea9" }, # 可以看到创建的数据卷宿主机路径 ... ...**

1. **直接指定获取**

**$ sudo docker inspect --format="{{ .Volumes }}" data**

**map[/data: /var/lib/docker/vfs/dir/151de401d268226f96d824fdf444e77a4500aed74c495de5980c807a2ffb7ea9]**

1. 挂载宿主机目录为一个数据卷

-v选项除了可以创建卷，也可以挂载当前主机的一个目录到容器中。

$ sudo docker run --name web -v /source/:/web -t -i ubuntu:14.04 /bin/bash

bash-4.1# ls -ld /web/

drwxr-xr-x 2 root root 4096 Jul 23 06:59 /web/

bash-4.1# df -Th

... ...

ext4 91G 4.6G 82G 6% /web

bash-4.1# exit

默认挂载卷是可读写的，可以在挂载时指定只读

$ sudo docker run --rm --name test -v /source/:/test:ro -t -i ubuntu:14.04 /bin/bash

## 6.2 创建和挂载一个数据卷容器

创建数据卷容器

$ sudo docker run -t -i -d -v /test --name test ubuntu:14.04 echo hello

使用--volumes-from选项在另一个容器中挂载 /test 卷。不管 test 容器是否运行，其它容器都可以挂载该容器数据卷，当然如果只是单独的数据卷是没必要运行容器的。

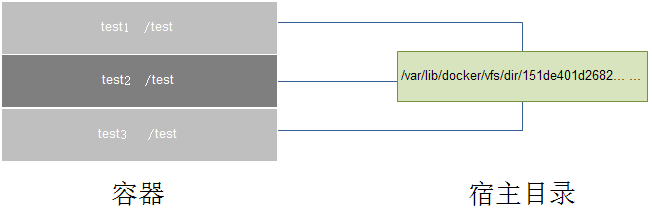
$ sudo docker run -t -i -d --volumes-from test --name test1 ubuntu:14.04 /bin/bash

添加另一个容器

$ sudo docker run -t -i -d --volumes-from test --name test2 ubuntu:14.04 /bin/bash

也可以继承其它挂载有 /test 卷的容器

$ sudo docker run -t -i -d --volumes-from test1 --name test3 ubuntu:14.04 /bin/bash



## 6.3备份、恢复或迁移数据卷

1. 备份

$ sudo docker run --rm --volumes-from test -v $(pwd):/backup ubuntu:14.04 tar cvf /backup/test.tar /test

tar: Removing leading `/' from member names

/test/

/test/b

/test/d

/test/c

/test/a

启动一个新的容器并且从test容器中挂载卷，然后挂载当前目录到容器中为 backup，并备份 test 卷中所有的数据为 test.tar，执行完成之后删除容器--rm，此时备份就在当前的目录下，名为test.tar。

$ ls # 宿主机当前目录下产生了 test 卷的备份文件 test.tar test.tar

1. 恢复

你可以恢复给同一个容器或者另外的容器，新建容器并解压备份文件到新的容器数据卷

$ sudo docker run -t -i -d -v /test --name test4 ubuntu:14.04 /bin/bash

$ sudo docker run --rm --volumes-from test4 -v $(pwd):/backup ubuntu:14.04 tar xvf /backup/test.tar -C /

# 恢复之前的文件到新建卷中，执行完后自动删除容器 test/ test/b test/d test/c test/a

## 6.4 删除 Volumes

Volume 只有在下列情况下才能被删除：

docker rm -v删除容器时添加了-v选项

docker run --rm运行容器时添加了--rm选项

否则，会在/var/lib/docker/vfs/dir目录中遗留很多不明目录。

# 七、链接容器

docker 允许把多个容器连接在一起，相互交互信息。docker 链接会创建一种容器父子级别的关系，其中父容器可以看到其子容器提供的信息。

## 7.1 容器命名

在创建容器时，如果不指定容器的名字，则默认会自动创建一个名字，这里推荐给容器命名：

1. 给容器命名方便记忆，如命名运行 web 应用的容器为 web
2. 为 docker 容器提供一个参考，允许方便其他容器调用，如把容器 web 链接到容器 db

可以通过--name选项给容器自定义命名：

$ sudo docker run -d -t -i --name test ubuntu:14.04 bash

$ sudo docker inspect --format="{{ .Nmae }}" test

/test

注：容器名称必须唯一，即你只能命名一个叫test的容器。如果你想复用容器名，则必须在创建新的容器前通过docker rm删除旧的容器或者创建容器时添加--rm选项。

## 7.2 链接容器

链接允许容器间安全通信，使用--link选项创建链接。

$ sudo docker run -d --name db training/postgres

基于 training/postgres 镜像创建一个名为 db 的容器，然后下面创建一个叫做 web 的容器，并且将它与 db 相互连接在一起

$ sudo docker run -d -P --name web --link db:db training/webapp python app.py

--link <name or id>:alias选项指定链接到的容器。

查看 web 容器的链接关系:

$ sudo docker inspect -f "{{ .HostConfig.Links }}" web

[/db:/web/db]

可以看到 web 容器被链接到 db 容器为/web/db，这允许 web 容器访问 db 容器的信息。

容器之间的链接实际是一个链接允许一个源容器提供信息访问给一个接收容器。在本例中，web 容器作为一个接收者，允许访问源容器 db 的相关服务信息。Docker 创建了一个安全隧道而不需要对外公开任何端口给外部容器，因此不需要在创建容器的时候添加-p或-P指定对外公开的端口，这也是链接容器的最大好处，本例为 PostgreSQL 数据库。

Docker 主要通过以下两个方式提供连接信息给接收容器：

* 环境变量
* 更新/etc/hosts文件

环境变量：

当两个容器链接，Docker 会在目标容器上设置一些环境变量，以获取源容器的相关信息。

首先，Docker 会在每个通过--link选项指定别名的目标容器上设置一个<alias>\_NAME环境变量。如果一个名为 web 的容器通过--link db:webdb被链接到一个名为 db 的数据库容器，那么 web 容器上会设置一个环境变量为WEBDB\_NAME=/web/webdb.

以之前的为例，Docker 还会设置端口变量:

$ sudo docker run --rm --name web2 --link db:db training/webapp env

. . .

DB\_NAME=/web2/db

DB\_PORT=tcp://172.17.0.5:5432

DB\_PORT\_5432\_TCP=tcp://172.17.0.5:5432

# <name>\_PORT\_<port>\_<protocol> 协议可以是 TCP 或 UDP

DB\_PORT\_5432\_TCP\_PROTO=tcp

DB\_PORT\_5432\_TCP\_PORT=5432

DB\_PORT\_5432\_TCP\_ADDR=172.17.0.5

. . .

注：这些环境变量只设置给容器中的第一个进程，类似一些守护进程 (如 sshd ) 当他们派生 shells 时会清除这些变量

更新/etc/hosts文件

除了环境变量，Docker 会在目标容器上添加相关主机条目到/etc/hosts中，上例中就是 web 容器。

$ sudo docker run -t -i --rm --link db:db training/webapp /bin/bash

root@aed84ee21bde:/opt/webapp# cat /etc/hosts

172.17.0.7 aed84ee21bde

. . .

172.17.0.5 db

/etc/host文件在源容器被重启之后会自动更新 IP 地址，而环境变量中的 IP 地址则不会自动更新的。

# 八、docker操作镜像

## 8.1 制作镜像

制作Docker镜像主要有如下两种方式：

1. 使用docker commit 命令来创建镜像

* 通过docker run命令启动容器
* 修改docker镜像内容
* docker commit提交修改的镜像
* docker run新的镜像

1. 使用 Dockerfile 来创建镜像

使用 docker commit 来扩展一个镜像比较简单，但是不方便在一个团队中分享。使用 docker build 来创建一个新的镜像。需要创建一个 Dockerfile，包含一些如何创建镜像的指令。

构建镜像的步骤:

1. 新建一个目录和一个 Dockerfile

$ mkdir new\_folder

$ cd new\_folder

$ touch Dockerfile

1. 编写Dockerfile，Dockerfile中每一条指令都创建镜像的一层，例如：

# 这里是注释

# 设置继承自哪个镜像

FROM ubuntu:14.04

# 下面是一些创建者的基本信息

MAINTAINER birdben (191654006@163.com)

# 在终端需要执行的命令

RUN apt-get install -y openssh-server

RUN mkdir -p /var/run/sshd

1. 示例

$ sudo docker build -t="birdben/ubuntu:v1" .

# 下面是一堆构建日志信息

############

我是日志

############

# 参数：

# -t 标记来添加 tag，指定新的镜像的用户和镜像名称信息。

# “.” 是 Dockerfile 所在的路径（当前目录），也可以替换为一个具体的 Dockerfile 的路径。

# 以交互方式运行docker

$ docker **run** -it birdben/ubuntu:v1 /bin/bash

# 运行docker时指定配置

$ sudo docker **run** -d -p 10.211.55.4:9999:22 ubuntu:tools '/usr/sbin/sshd' -D

# 参数：

# -i：表示以“交互模式”运行容器，-i 则让容器的标准输入保持打开

# -t：表示容器启动后会进入其命令行，-t 选项让Docker分配一个伪终端（pseudo-tty）并绑定到容器的标准输入上

# -v：表示需要将本地哪个目录挂载到容器中，格式：-v <宿主机目录>:<容器目录>，-v 标记来创建一个数据卷并挂载到容器里。在一次 run 中多次使用可以挂载多个数据卷。

# -p：指定对外80端口

# 不一定要使用“镜像 ID”，也可以使用“仓库名:标签名”

## 8.2更新镜像

使用commit更新镜像库指定镜像

$ sudo docker commit 614122c0aabb aoct/apache2